Are Code Smell Co-occurrences Harmful to Internal Quality Attributes? A Mixed-Method Study

Júlio Martins
Campus Quixadá - UFC
Quixadá, CE, Brazil
juliomserafim@gmail.com

Carla Bezerra
Campus Quixadá - UFC
Quixadá, CE, Brazil
carlailane@ufc.br

Anderson Uchôa
DI - PUC-Rio
Rio de Janeiro, RJ, Brazil
auchoa@inf.puc-rio.br

Alessandro Garcia
DI - PUC-Rio
Rio de Janeiro, RJ, Brazil
afgarcia@inf.puc-rio.br

ABSTRACT

Previous studies demonstrated how code smells (i.e., symptoms of the presence of system degradation) impact the software maintainability. However, few studies have investigated which code smell types tend to co-occur in the source code. Moreover, it is not clear to what extent the removal of code smell co-occurrences – through refactoring operations – has a positive impact on quality attributes such as cohesion, coupling, inheritance, complexity, and size. We aim at addressing these gaps through an empirical study. By investigating the impact of the smells co-occurrences in 11 releases of 3 closed-source systems, we observe (i) which code smells tend to co-occur together, (ii) the impact of the removal of code smell co-occurrences on quality internal attributes before and after refactoring, and (iii) which are the most difficult co-occurrences to refactoring from the developers' perspective. Our results show that 2 types of code smell co-occurrences generally tend to co-occur. Moreover, we observed that the removal of code smells co-occurrences lead to a significant reduction in the complexity of the systems studied was obtained. Conversely, cohesion and coupling tend to get worse. We also found that two code smells co-occurrences (*God Class–Long Method* and *Disperse Coupling–Long Method*) as the most difficult to refactor indicating that attention is needed not to insert these anomalies in the source code. Based on our findings, we argue that further research is needed on the impact of code smells co-occurrences on internal quality attributes.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; **Maintaining software**.

KEYWORDS

Code Smells Co-occurrences. Refactoring. Quality Attributes.

ACM Reference Format:
Júlio Martins, Carla Bezerra, Anderson Uchôa, and Alessandro Garcia. 2020. Are Code Smell Co-occurrences Harmful to Internal Quality Attributes? A Mixed-Method Study. In *34th Brazilian Symposium on Software Engineering (SBES '20), October 21–23, 2020, Natal, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3422392.3422419

1 INTRODUCTION

The software quality is critical and essential in different types of organizations and systems, such as real-time systems and control systems. A good quality of software can allow the evolution of these systems to have less effort and financial expenses [3]. Throughout its evolution, the software systematically undergoes changes that can lead to the deterioration of the quality of its structure [5, 49]. In this context, the concept of code smells arises, which are anomalous code structures that represent symptoms that affect the maintainability of systems at different levels, such as classes and methods [20, 27]. Code Smells can indicate problems related to aspects of code quality such as understandability and modifiability [20].

Some studies have evaluated not only the individual occurrences of code smells, but also the relationships between these anomalies and the impact they have on the software quality [55, 56]. The presence of individual occurrences of code smells does not significantly affect the understanding of the software or the performance of the developers, the opposite happens when there is a co-occurrence of these anomalies [1]. Pietrzak and Walter [41] were the first to investigate the relationship between code smells, which they call inter-smell relations. The authors define six types of code smells co-occurrence and claim that the study of these associations and dependencies between code smells can result in a better understanding of potential problems in software quality.

Yamashita and Moonen [55] and Sjøberg et al. [44], indicate that the code smell co-occurrence is not good indicators of maintainability. Oizumi et al. [37] analyzed the code smells co-occurrences to identify architectural problems: they suggest that clusters of smells are significantly better indicators of code problems than individual instances of smells. A possible relationship between Duplicated Code and Long Method code smells was found by Martins et al. [32]. The researchers found that the number of Duplicated Code occurrences increases as the number of Long Method occurrences also increases. de Paulo Sobrinho et al. [12] conducted a literature review on code smells. As a result of study [12], the authors identified that code smells co-occurrences can cause maintenance and design problems and that more empirical studies are needed to investigate the impact of these anomalies on the source code.

Refactoring is a process of improving software systems by applying code transformations [20] as a means to achieve various developers' intents and goals [38]. Refactorings can remove code smells and co-occurrence between them, and this can have a direct impact on code quality [54]. However, the systematic review performed by Lacerda et al. [26] indicates that there are many opportunities to use refactoring to remove code smells and their co-occurrences, and it is a major research challenge to indicate which refactoring strategies should be applied.

Despite the large number of studies investigating the effects of individual occurrences of code smells [43], there are still few studies that investigate the effects of co-occurrences between code smells. Most of these studies investigate the effects of the co-occurrence of code smells in open source projects [19, 22, 39]. As it is still a recent area, studies focus on: investigation of the effects between code smell co-occurrences and code maintainability [17, 32, 55], detection of code smells co-occurrences [18, 22, 39–41, 56] and correlation analysis between co-occurrences [52]. Therefore, further investigation is needed on impact code smells co-occurrences on quality in closed-source systems [11].

Having established and substantiated the importance of studying the influence of code smells co-occurrences for software quality in the context of object-oriented systems. We aim to conduct a study to investigate the impact of code smells co-occurrences on three OO closed-source systems in internal quality attributes of these systems. To guide our study, we have elaborated three research questions that we want to investigate:

RQ*$_1$: *Which are the most frequent code smells co-occurrences in closed-source projects?

RQ*$_2$: *How does the removal of code smells co-occurrence affect internal quality attributes in closed-source projects?

RQ*$_3$: *Which code smells co-occurrences are considered as the most difficult to remove from the developer's perspective?

From these research questions, our study obtained the following results:

- The most frequent co-occurrences of code smells across projects are *God Class–Long Method* and *Disperse Coupling–Long Method.*
- Co-occurrences increase during the development of the system and that the removal of these anomalies has a negative impact on the cohesion and coupling attributes.
- The removal of code smells co-occurrences suggests a significant decrease in the complexity of the systems.
- *God Class - Long Method* and *Disperse Coupling - Long Method* co-occurrences were considered the most difficult to refactor by developers.

The remainder of this paper is organized as follows. Section 2 presents the theoretical background to support the understanding of this study. Section 3 describes the research method. Section 4 presents the results of the study. Section 5 discusses threats to the study. Section 6 compares this study with previous studies. Finally, Section 7 presents the conclusions and future work.

2 BACKGROUND

2.1 Code Smells Detection

Code smells can indicate problems related to aspects of code quality. As a consequence, they can cause problems for developers in activities in the maintenance phase of the software [20]. These indicators can affect any system [30, 36]. Previous work [11, 30, 36, 49] provide evidence that code smells are sufficient indicators of parts of code affected by poor feature decomposition in the case of single software systems.

There are several tools for detecting code smells [16]. This study will use the JSpIRIT tool, a tool with a semi-automatic approach that focuses on the most critical code smells in the system [50]. Currently this tool can detect 10 code smells [51]. In our study we detected the following code smells using this tool: Feature Envy, Disperse Coupling, Intensive Coupling, Refuse Parent Bequest, Shotgun Sugery. Another tool used in our study is JDeodorant, which is an Eclipse plugin for detecting code smells [47]. Currently five code smells are supported by the tool: Long Method, Feature Envy, God Class, Type/State Checking, Duplicated Code [47]. In this work, this tool will detect the God Class and Long Method code smells. We choose these tools due to their availability and wide use in previous work [32, 48]. In our study, we studied smells with different levels of granularity and, we focused on the smell types with the highest frequencies across the projects, for that reason 7 types of code smells were analyzed, which are described in Table 1.

Table 1: Code smells analyzed by this work [20]

Code Smells	Description
Feature Envy	Instructions for a method that should be moved to another method, sometimes located in another class, whose features are more shared and used
God Class	A very large and complex class, which usually concentrates a lot of software functionality.
Disperse Coupling	Method that calls one or more methods of several classes
Intensive coupling	Method that calls several methods from other classes
Refused parent bequest	Subclass that does not use the protected methods of its superclass
Shotgun surgery	Method called by many methods from other classes
Long Method	Methods with many lines of code and a lot of software logic

2.2 Code Smell Co-occurrences

Code smells co-occurrences occur when there are relationships and dependencies between two or more code smells. For example, the same class that is God Class and also has a Duplicated Code [41].

Before detecting code smell co-occurrences, you must first detect individual code smells. This is necessary to verify and map the co-occurrences of code smells (i.e. the appearance of more than one code smell in the same method or the appearance of code smells in the same class). Thus, our study analyzed the co-occurrences of seven code smells.

There are several studies in the literature that study only individual instances of code smells [1, 24, 44]. However, few studies address or analyze code smell co-occurrences. Thus, more empirical studies are needed considering these relations [56]. Pietrzak and Walter [41] defines some types of code smell co-occurrences aiming at greater precision in the detection of code smells and the negative impact that these anomalies can cause in systems.

After detecting individual occurrences of code smells, it is possible to verify the co-occurrences of code smells at the method level and at the class level, at the method level they occur when there are two or more code smells in a given method (i.e., *Feature Envy–Long Method*). A class level happens when there is a class-level code smells (i.e., God Class) along with some other code smells (i.e., Long Method) [20]. Lanza and Marinescu [27] identified some relationships between code smells co-occurrences using the keywords: has/use. For example, God Class has Disperse Coupling (class level) and Intensive Coupling uses Shotgun Surgery (method level).

Table 2 shows examples of code smells co-occurrences at the class and method level. In the first example there is a code smell co-occurrence (*Long Method and God Class*), that is, Class1 which is a God Class has Method1 which is Long Method. In the second example, there is a co-occurrence at the method level in which the two code smells Long Method and Feature Envy are "together" in Method2. This example represents how we identified the code smells co-occurrences in our study [41].

Table 2: Examples of code smells co-occurrences

Class	Method	LM	FE	GC
Class1	method1()	X		X
Class2	method2()	X	X	

In this way, individual occurrences of code smells were detected by automatic tools, and co-occurrences were identified manually by us so that the refactoring process starts. Refactoring consists of changing the source code of a system. It can improve the internal structure by improving the measures of internal quality attributes such as cohesion, coupling, complexity, size and inheritance [20]. With the refactoring of code smells performed by the developers to remove co-occurrences, we want to investigate whether there is any impact on the internal quality attributes of the system.

2.3 Internal Quality Attributes Measures

The ISO/IEC 25010/2011 standard [21], defines the quality as the degree to which the system satisfies what was established and the implicit needs of its stakeholders. Software quality can be measured by different quality attributes, which can be classified as: (i) internal quality attributes and (ii) external quality attributes [31].

External quality attributes are those that indicate the quality of the system based on factors that consider the software environment and the interactions between that environment and the software artifacts. An example of an external quality attribute is maintainability, which depends on a set of external factors to be evaluated, such as: the system's lifetime and the environment for which the system is being modified [2].

Internal quality attributes, such as size, cohesion and coupling, are those that can only be measured using only software artifacts. Quantifying internal quality attributes is much easier than quantifying external attributes, for instance, the size of a class can be measured using the LOC metric [34]. In our study, we used 13 metrics of internal quality attributes (see Table 3) well known in the literature [8, 13, 28, 33]. The metrics proposed by Chidamber and Kemerer [8] are pioneers in the area of object-oriented metrics and have a theoretical basis for measuring OO code. In this study, *CK metrics* [8] was one of the ones chosen to be used to check the internal quality of systems [14, 31, 48]. To collect the metrics we use the Understand tool to measure all internal quality attributes [7].

Table 3: Metrics of the internal quality attributes analyzed in this work [8, 13, 28, 33]

Attributes	Metric	Description
Cohesion	Lack of Cohesion of Methods (LCOM2) [8]	Measures cohesion of a class. The higher the value of this metric, less cohesive is the class.
Coupling	Coupling Between Objects (CBO) [8]	Number of classes that a class is coupled. The higher the value of this metric, more coupling is the classes and methods.
Complexity	Average Cyclomatic Complexity (ACC) [33]	Average cyclomatic complexity of all nested methods. The higher the value of this metric , more complex is the classes and method.
	Sum Cyclomatic Complexity (SCC) [33]	Sum of cyclomatic complexity of all nested methods. The higher the value of this metric, more complex is the classes and methods.
	Nesting (MaxNest) [28]	Maximum nesting level of control constructs. The higher the value of this metric, more complex is the classes and methods.
	Essential Complexity (EVG) [33]	Measure of the degree to which a module contains unstructured constructs. The higher the value of this metric, more complex is the classes and methods.
Inheritance	Number Of Children (NOC) [8]	Number of subclasses of a class. The higher the value of this metric greater is the degree of inheritance of a system.
	Depth of Inheritance Tree (DIT) [8]	The number of levels that a subclass inherits from methods and attributes of a superclass in the inheritance tree. The higher the value of this metric greater is the degree of inheritance of a system.
	Bases Classes (IFANIN) [13]	Immediate number of base classes. The higher the value of this metric greater is the degree of inheritance of a system.
Size	Lines of Code (LOC) [28]	Number of lines of code excluding spaces and comments. The higher the value of this metric the larger the system size.
	Lines with Comments (CLOC) [28]	Number of lines with comment. The higher the value of this metric the larger the system size.
	Classes (CDL) [28]	Number of classes. The higher the value of this metric the larger the system size.
	Instance Methods (NIM) [28]	Number of instance methods. The higher the value of this metric the larger the system size.

3 STUDY SETTINGS

3.1 Goal and Research Questions

The study of co-occurrences and relations between code smells is a relatively new area of research. Although there are already studies in the literature on this topic [39, 41, 52, 56]. None of theses studies analyzed the impact of co-occurrences or relationships between code smells on internal quality attributes. Due to limited empirical knowledge on this subject, our study aims to investigate the impact of code smells co-occurrences to internal quality attributes.

We summarize our study goal as follows [53]: (i) *analyze* the code smells co-occurrences; *for the purpose of* understating their impact on internal attributes of software quality; *with respect to* which code smells tend to co-occur together, (ii) the removal of code smell co-occurrences before and after software refactoring, and (iii) which are the most difficult co-occurrences to refactoring; *from the viewpoint of* researchers and software developers; *in the context of* three closed-source systems. Our research questions (RQs) are discussed as follows.

RQ$_1$*: *Which are the most frequent code smells co-occurrences in closed-source projects? – **RQ$_1$** aims at identifying which and how often code smells tend to co-occur together. By answering **RQ$_1$**, we can reveal the existence of different patterns of code smells co-occurrences during the process of software development. Additionally, we can reveal insights for new research in which the study of these co-occurrences of smells has not yet been carried out.

RQ$_2$*: *How does the removal of code smells co-occurrence affect internal quality attributes in closed-source projects? – **RQ$_2$** aims at providing evidence on the impact of the removal of co-occurrence of code smells on internal quality attributes. Differently from previous studies [17, 36] that tend to investigate the introduction of co-occurrence of code smells, **RQ$_2$** assesses the impact of the removal of co-occurrence of code smells on five internal quality attributes (software cohesion, coupling, complexity, inheritance, and size). To achieve this goal, the removal of co-occurrences was carried out in practice with the developers of the systems analyzed. By answering **RQ**$_2$, we can reveal how the removal of these co-occurrences impacts on the internal quality attributes.

RQ**$_3$**: Which code smells co-occurrences are considered as the most difficult to remove from the developer's perspective? – **RQ**$_3$ assesses which are the most difficult co-occurrences to refactor from the point of view of the project developers. Our objective with this research question is to list which are the main co-occurrences of code smells that developers should be careful not to insert into the code during the development process.

3.2 Study Steps

This section describes the study steps, in order to support the investigation of code smells co-occurrences.

Step 1: Select software systems for analysis. We selected 3 Java closed-source software systems that are being developed by our industrial partners. For this purpose, we ask project managers to refer us to projects with the following criteria: (i) systems with the most lines of code; (ii) Systems that were not in their initial versions; (iii) systems written in the Java language; and (iv) systems that are already in a production environment. Table 4 presents general data per system. The first column names the system[1]. The remainder columns present: system domain; number of classes; number of releases; and, number of lines of code (LOC). We collected all data via Understand.

Table 4: General data of the target software systems

System	Domain	# of classes	# of releases	# LOC
S1	Electronic dental record	145	5	7830
S2	Academic offer	99	4	5623
S3	Warehouse	106	3	5447

The **S1** aims to provide integrated monitoring of patients seen at the different dentistry clinics. **S2** aims to assists in the process of offering required courses at the beginning of each academic semester at the University. Finally, **S3** aims to manage warehouses at the dental clinics of the University, allowing to control the stock of materials used in the Dentistry course, in addition to making material inputs and outputs individually for each clinic. All the target systems are web-based and developed using Spring Boot, Thymeleaf and Jquery technologies.

Step 2: Identify code smells and their co-occurrences. We identified seven types of code smells: *Feature Envy*, *God Class*, *Disperse Coupling*, *Intensive Coupling*, *Refused Parent Bequest*, *Shotgun Surgery* and *Long Method.* Table 1 describes the seven code smells collected. The code smells were collected using two tools, JDeodorant [24] and JSpIRIT [50]. Next, we identified the co-occurrences of the code smells collected. The types of relationships used to identify the co-occurrences are described in Section 2.2. This step is replicated for all releases of the three selected software systems. Therefore, it will be possible to identify which code smells co-occurrences tend to appear most during the development process and the total number of these relationships.

[1]We omitted their names due to intellectual-property constraints.

Step 3: Measure internal quality attributes. Table 3 presents the 13 code metrics that we used to measure internal quality attributes [8, 13, 28]. The first column lists the internal quality attributes. The second column presents the software metrics related to each internal quality attribute. Finally, the third column describes each metric. To compute each metric, we used a non-commercial license of the Understand tool. We selected theses metrics because they enable us to assess different properties of each attribute [6, 8], such as LOC and CBO that measures the size and coupling, respectively. Therefore, these code metrics can reveal the effect of code smells co-occurrences on these internal quality attributes. We chose to perform the analysis in a class-level scope. In total we measurement five quality internal attributes: cohesion, coupling, complexity, size, and inheritance.

Step 4: Removal of code smells co-occurrences with software developers. This step aims to conduct the removal of co-occurrences of the code smells identified in Step 2. For this purpose, we have recruited developers who contributes to the development of each selected software system to participate as subjects in the study. Thus, we sent a *Characterization Form* for each developer. This form aimed to characterize the developer regarding education, experience with software development, and their projects. Their answers were analyzed to determine which of them were eligible to participate in the study. Table 5 summarizes the characteristics of each developer selected for the experiment. All the developers are from the same company, but not everyone was aware of all systems; 5 had prior knowledge of S1, 4 of S2, and 5 of S3. The company released the developers as a regular part of the job.

Table 5: Characterization of developers

ID	Experience in years	Education Level	Quality Metrics	Code Smells	Java
P1	5	Graduate Degree	Advanced	Intermediary	Intermediary
P2	1	Graduate Degree	Basic	Basic	Intermediary
P3	2	Graduate Degree	Advanced	Advanced	Advanced
P4	5	Graduate Degree	Intermediary	Basic	Advanced
P5	2	Graduate Degree	Intermediary	Intermediary	Intermediary
P6	3	Graduate Degree	Advanced	Advanced	Advanced
P7	5	Master Degree	Intermediary	Intermediary	Intermediary

After selected the developers we asked them to perform the removal of code smells co-occurrences (method level and class level) in their systems thought manual software refactoring. We explain in more detail the experimental procedure used to remove the code smells co-occurrences in Section 3.3.

Step 5: Perform a new measurement of the internal quality attributes. After the removal of code smells co-occurrences, we performed new measurements of the internal quality attributes. Our goal was to compare the value of the metrics for each release of the system before and after the removal of the code smells co-occurrences thought manual software refactoring. The set of quality metrics used to measure the internal quality attributes are described in Table 3. The comparison was made through the results of the quality metrics, and the comparison it was done with the most current version of each system before removing the co-occurrences and with that same system after the procedure of removing code smell co-occurrences.

To assess whether the quality of the systems improved or worsened after removing code smells co-occurrences, we used the same approach of Tarwani and Chug [46], in which the authors use the sum of the metrics to compare the quality of systems. This means that if the value of the sum of the metrics of each internal quality attribute increased that internal quality attribute had a worsening, for example we used four metrics to evaluate the complexity attribute as shown in the Table 3 we measure and calculate the sum of the values of each metric of this attribute before removing co-occurrences and after refactoring three scenarios are considered:

(1) If the sum of these metrics has decreased, then complexity has decreased.
(2) If the sum of these metrics has increased, then the complexity has increased.
(3) If there was no difference between the sums then the complexity has not changed.

In this way, we take this approach to all other metrics and internal quality attributes. Details on detecting code smell co-occurrences and measuring systems before and after removing code smells co-occurrences are found in our research website[2].

3.3 Experimental Procedures

The study was composed by a set of four activities. We describe each activities as follows.

Activity 1: Training session. We conducted a *training session* with all participants about essential concepts for the study, such as code smells, internal quality attributes, and refactoring. We also trained the participants about how to identify the code smells co-occurrences. We spent an hour and a half. We presented a set of practical examples that illustrate refactoring operations that could be applied in each code smell presented in the first part of the training. Next, we provide a set of toys examples for developers to apply refactoring methods to remove code smells. We spent two hours. We decided to provide a training session to level up their knowledge about the main concepts regarding our study. Thus, we tried to reduce the bias by focusing on main concepts and presenting theoretical and practical examples.

Activity 2: Removal of co-occurrences of code smells via manual refactoring. We asked developers to perform the removal of code smells co-occurrences (method level and class level) in their systems thought manual software refactoring. For instance, if there is an occurrence of the Long Method and Feature Envy in the same method, the developer can choose only one of these code smells to remove. This removal results in the elimination of the co-occurrence at the method level. On the other hand, a co-occurrence at class-level happens when there is a God Class or Refused Parent Bequest along with some other code smell, in which case the developers could also choose which of the code smells to removal, thus de-characterizing the class-level co-occurrence.

To support the removal of code smells co-occurrences, we provided participants a list that summarized the name of methods or classes in which the co-occurrences of code smells were identified from Step 2. Additionally, for each code smell co-occurrence, we created issues on the Github related to refactoring activities. Each issue contained information about the class and the method affected by a code smell. Thus, the developers were free to choose issues, and consequently, which code smell to refactor to remove the co-occurrence. We conducted weekly meetings to check the progress of the activities and if the developers founded any type of difficulty or obstacle in the refactoring process. We instructed developers to make it clear which commits were related to a refactoring activity. Thus, each commit has tagged to with the label representing the name of the code smell to be refactored to de-characterize the co-occurrence (i.e. God Class). Additionally, separate branches were created for each of the refactoring activities.

Activity 3: Validation of the removal of code smells co-occurrences. We analyzed the commits to see if the co-occurrence was eliminated by the programmer. At each refactoring commit, we validate the refactoring of the co-occurrence. If validated, the branch code should be committed to the master repository. If the activity was not validated, the developer should refactor one more time until the co-occurrence is no longer characterized.

The analysis of the projects occurred at different times, that is, we did not analyze the three systems at the same time. The first project we studied was the S1 system, the second was the S2 system and finally the S3 system. The entire refactoring, analysis and study process for the three projects took three months and included seven different developers.

And to perform the removal of code smells co-occurrences in each system, several commits were necessary. As can be seen in Table 6. In this table we present the number of refactoring commits and the number of co-occurrences removed in each of the systems.

Table 6: Number of refactoring commits and number of co-occurrences removed

System	# co-occurrences removed	# refactoring commits	# total of commits
S1	37	95	2993
S2	33	51	1045
S3	24	37	1217

Activity 4: Apply follow-up questionnaire. After refactoring activities, we apply a questionnaire to check the developers' perception of these activities. For example, we asked them if they had already used the concepts of code smells, refactoring and quality metrics. We also asked to tell us which were the hardest and easiest code smells and the hardest and easiest co-occurrences to refactor and we asked the reasons for their responses.

4 RESULTS AND DISCUSSION

4.1 The Frequency of Code Smells Co-occurrences (RQ_1)

We address $\mathbf{RQ_1}$ by identifying the most frequent co-occurrence of code smells in the releases of the three target systems. The procedures that we used to identify the code smells co-occurrences are described in Section 3.3. Table 7 presents the frequency of each co-occurrence grouped by system and release. The first and second columns list each system and release, respectively. The remaining columns present each code smell co-occurrence.

The most frequent code smells co-occurrences. Table 7 allows us to observed that there are at least five types of code smells

[2]https://julioserafim.github.io/SBES2020/

Table 7: Code smells co-occurrences in the three systems

System	Release	FE and LM	DCO and LM	DCO and FE	GC and LM	IC and LM	GC and SS	FE and GC	FE and RPB	DCO and GC	FE and SS
S1	v1.0	9	1	5	7	0	1	2	2	0	0
	v1.1	15	6	4	7	1	1	5	2	0	0
	v1.2	10	8	3	9	1	1	4	2	0	0
	v1.3	12	8	3	9	1	1	3	0	0	0
	v1.3.1	11	8	3	7	1	1	3	3	0	0
	Total	**57**	**31**	**18**	**39**	**4**	**5**	**17**	**9**	**0**	**0**
S2	v0.1	2	2	3	1	1	6	4	0	5	0
	v0.2	3	4	4	6	0	0	1	0	5	0
	v0.2.1	3	5	4	8	1	4	2	0	6	0
	Total	**8**	**11**	**11**	**15**	**2**	**10**	**7**	**0**	**16**	**0**
S3	v0.1	1	2	2	5	0	0	1	0	2	0
	v0.2	1	1	0	2	0	0	2	0	1	0
	v1.0	3	4	1	6	0	2	5	0	2	1
	Total	**5**	**7**	**3**	**13**	**0**	**2**	**8**	**0**	**5**	**1**

that tend to compose a co-occurrence: *Feature Envy*, *Long Method*, *Disperse Coupling*, *God Class*, and *Shotgun Surgery*. Regarding the most frequent types of code smells co-occurrence, we have some interesting observations. The ranking of the top-5 co-occurrence of code smells per system, from the most frequent to the less frequent indicate that *God Class–Long Method* and *Disperse Coupling–Long Method*, are the co-occurrences that most tend to co-occur together.

Table 8: Co-occurrences that most tend to co-occur

Co-occurrences	# of systems that co-occur	Systems
God Class–Long Method	3	S1,S2,S3
Disperse Coupling–Long Method	3	S1,S2,S3
Feature Envy–Long Method	2	S1,S3
Disperse Coupling–Feature Envy	2	S1,S2
Feature Envy–God Class	2	S1,S3

Moreover, we also found code smells co-occurrences that are the most detected in at least two systems, such as: *Feature Envy–Long Method* in the S1 and S3 systems, *Disperse Coupling–Feature Envy* in the S1 and S2 systems and *Feature Envy–God Class* in the S1 and S3 systems, indicating a pattern that these co-occurrences tend to co-occur. Table 8 shows the co-occurrences that most tend to co-occur in the systems studied. These results confirm what was found in previous work in the literature on code smells co-occurrences [27, 29, 39, 55].

Finding 1: The most frequent co-occurrences of code smells across projects are *God Class–Long Method* and *Disperse Coupling–Long Method.*

Code smells co-occurrences tend to increase during software evolution. By comparing the first and the last release of each system, we can observe an increase in the number of code smells co-occurrences in most of the analyzed systems. More precisely, for the S1 system we observed an increase of five out of eight (**62.5**%) co-occurrences: *Feature Envy–Long Method*, *Disperse Coupling–Long Method*, *Intensive Coupling–Long Method*, *Feature Envy–God Class*, and *Feature Envy–Refused Parent Bequest*. In the case of S2 system, we also observed an increase of (**62.5**%) in the following co-occurrences: *Feature Envy–Long Method*, *Disperse Coupling–Long Method*, *Disperse Coupling–Feature Envy*, *God Class–Long Method* and *Disperse Coupling–God Class*. Finally, in S3 system, an increase in 6 out of 8 (**75%**) in the following code smells co-occurrences was observed: *Feature Envy–Long Method*, *Disperse Coupling–Long Method*, *Disperse Coupling–Feature*, *God Class–Long Method*, *God Class–Shotgun Surgery*, *Feature Envy–God Class*, and *Feature Envy–Shotgun Surgery*.

These results suggest that co-occurrences tend to increase over time. One of the factors that can explain this phenomenon is the number of features in each release. The latest releases have more features than the first. We have an understanding that developers should also be concerned with the number of individual instances of code smells like *Long Method*, *God Class* and *Feature Envy* because these code smells tend to co-occur with some other code smell. In Table 7, it is possible to notice that each of these code smells are present in at least four co-occurrence relations. However, they are necessary more empirical studies to verify the relationship between the number of features and the number of code smells co-occurrences.

Finding 2: Most co-occurrences increased over the systems development.

4.2 The Impact of the Removal of Code Smells Co-occurrences (RQ$_2$)

We address **RQ**$_2$ by evaluating the impact of the removal of code smells co-occurrences on five internal quality attributes: cohesion, inheritance, size, coupling, and complexity. We emphasize that the removal of co-occurrence was preformed in practice through applied manual refactorings by developers familiar with the systems (see Section 3.2). Table 9 presents the impact of removing code smells co-occurrences for the internal quality attributes, considering the latest releases of the three target systems.

We put the quality attributes and their respective metrics. We analyzed the quality of the three systems using the Understand tool before the process of removing code smells co-occurrences and the computed value for each metric can be seen in the Table 9.

We identified that after removing code smells co-occurrences, cohesion decreased in the three systems studied by us in this work, coupling and inheritance increased in the three systems, complexity decreased significantly in two systems (S1 and S3) and increased

Table 9: Impact of the removal of co-occurrences of code smells grouped by the system and internal quality attribute

System		Cohesion	Complexity				Inheritance			Coupling	Size			
		LCOM2	**ACC**	**SCC**	**EVG**	**MaxNet**	**DIT**	**NOC**	**IFANIN**	**CBO**	**LOC**	**CLOC**	**NIM**	**CDL**
S1 with co-occurrences		3596	103	1120	867	111	172	34	138	551	7830	166	919	146
	Total	3596				2201			344	551				9061
S1 without co-occurrences		3878	110	1223	247	116	186	35	151	536	8416	199	1004	163
	Total	3878				1696			372	536				9782
Results		**↓7.75%**			**↓22.94%**			**↑ 8.13%**		**↑3.2%**		**↑7.95%**		
S2 with co-occurrences		3300	101	881	177	74	111	15	119	332	7094	112	719	106
	Total	3300				1233			245	332				8031
S2 without co-occurrences		3438	102	913	175	74	113	16	124	333	5623	109	748	112
	Total	3438				1264			253	333				6592
Results		**↓4.1%**			**↑2.5%**			**↑3.2%**		**↑0.3%**		**↓17.91%**		
S3 with co-occurrences		3634	86	770	102	62	101	12	109	313	5082	151	640	99
	Total	3634				1020			222	313				5972
S3 without co-occurrences		3856	78	579	93	53	108	11	119	334	5447	126	696	104
	Total	3856				803			238	334				6373
Results		**↓6.1%**			**↓21.27%**			**↑ 7.2%**		**↑ 9.9%**		**↑6.71%**		

in one system (S2). The size of the S1 and S3 systems has slightly increased and the size of the S2 system has been reduced.

We identified that the cohesion of the S1 system worsened by **7.75%** after removing of code smells co-occurrences, this can be seen through the value of the LCOM2 metric that with code smells co-occurrences was 3596 and after the process of removing co-occurrences went to 3878. The higher the value of this metric, the worse is the cohesion of a system [8]. On the other hand, the value of the complexity metrics decreased after removing co-occurrences. To evaluate the impact of removing code smells on attributes with more than one metrics, we compared the sum of the metrics value [46] before removing code smells and after removing code smells, as shown in the Table 9. We can verify in the Table 9 that the sum of the metrics of complexity attribute decreased from 2021 to 1696, indicating a decrease of **22.94%** of the total complexity value [8, 35].

In the S1 system, the inheritance and size attributes obtained an increase of **8.13%** and **7.95%** respectively in the value of their metrics. The coupling increased by **3.2%**, it is expected that there was a reduction in the percentage coupling [9]. A possible explanation for the coupling to have increased, despite the removal of co-occurrences, is the increase in the value of the inheritance metrics since it was previously found that the increase in inheritance may mean an increase in the coupling of classes [25].

Te S2 system, was the only system studied in this work that obtained a great decrease in its size after the removal of code smells co-occurrences. As we can see in the Table 9 the size decreased by **17.91%**. The number of lines of code (LOC) decreased from 7094 to 5623, but the number of methods (NIM) and the number of classes (CDL) increased. Although the complexity attribute did not decrease in this system, there was a small increase of **2.5%** in complexity and also yielded the lowest reduction in the cohesion and lower Coupling metrics and increased value in the attribute inheritance. Finally, in the S3 system, we identified a decrease of **6.3%** in the cohesion attribute, an increase in inheritance of **7.2%** and in coupling of **9.9%**, indicating once again that the increase in inheritance in the system may suggest an increase in coupling. The size also increased by **6.71%**. However, there was a significant decrease in complexity by **21.27%**.

The negative impact of removing code smells co-occurren -ces. Removing co-occurrences from code smells had a negative impact on internal quality attributes such as cohesion and coupling. After removing these anomalies, we found that these attributes worsened in all three systems studied in our work.

Finding 3: The removal of code smells co-occurrences did not have a positive impact on attributes such as cohesion and coupling.

The positive impact of removing code smells co-occurren -ces. On the other hand, the removal of code smells co-occurrences has managed to significantly reduce the complexity in S1 and S3 systems. Several work in the literature have already studied the impact of complexity for OO systems, most of these studies associate complexity with problems such as worsening software maintainability, greater propensity for errors and quality reduction [4, 10, 35, 45]. In the S1 system, complexity has been reduced by 22.94% and in the S3 system it has been reduced to 21.97%.

Finding 4: After removing the co-occurrences, the complexity decreased 22.94% and 21.27% respectively in two systems of the three target systems studied. Indicating that complexity can is an attribute that decreases with the removal of co-occurrences.

The data found in this research question suggest evidence that the removal of code smells co-occurrences can mean a reduction in the complexity attribute. However, more empirical studies need to be carried out to gain a better understanding.

4.3 The Code Smells Co-occurrences most Difficult to Remove (RQ_3)

We address $\mathbf{RQ_3}$ by asking developers to answer a questionnaire about refactoring activities. In this questionnaire, we asked the developers to inform on a scale of 1 to 5 which were the code smells co-occurrences most difficult to refactor, where 1 is the easiest and 5 the most difficult, if the developer had not refactored a certain co-occurrence he checked the option "*I did not refactor this code smell co-occurrence*". In addition, we also asked developers to write in a field the reasons for these choices so that we could have a better understanding of developers' perceptions.

We organize the data in a ranking, for that, we used the *Borda count* technique [42]. The *Borda count* is a ranking technique designed to obtain a consensus instead of a majority. We used this technique as follows. If they have n candidates, the first in the ranking has n points, the second n-1 points, the third one has n-2 points, and so on. This technique was also used in a previous study that ranked code smells more popular among developers [54]. Table 10 present the ranking of code smells co-occurrences that are more difficult to refactor.

Table 10: Co-occurrences more difficult to refactor

Position	Code smell co-occurrence	Points
1	*God Class–Long Method*	61
	Disperse Coupling–Long Method	61
2	*Feature Envy–Long Method*	57
	Feature Envy–God Class	57
3	*Disperse Coupling–Feature Envy*	53
4	*God Class–Shotgun Surgery*	50
	Feature Envy–Shotgun Surgery	50
5	*Feature Envy–Refused Parent Bequest*	49
	Disperse Coupling–God Class	49
6	*Intensive Coupling–Long Method*	47

The developer's perspective on code smell co-occurrence. We observed that under the developer's perspective that the co-occurrences most difficult to refactor from the most frequent to the less frequent: (1) *God Class–Long Method* and *Disperse Coupling–Long Method*; (2) *Feature Envy–Long Method* and *Feature Envy–God Class*; (3) *Disperse Coupling–Feature Envy*; and *Intensive Coupling–Long Method* co-competition with a significant difference of points in relation to the co-occurrences that are at the top.

Finding 5: *God Class–Long Method* and *Disperse Coupling–Long Method* co-occurrences were the most difficult to remove from the developers' point of view.

With this result we identified a potential problem involving code smells co-occurrences. In the Section 4.1, we found that the *God Class–Long Method* and *Disperse Coupling–Long Method* co-occurrences were the most likely to co-occur. However, as shown in the Table 7, these were also the most difficult co-occurrences to refactor. This finding suggests that developers should take care not to insert these co-occurrences in the source code.

Finding 6: Developers should be careful not to insert *God Class–Long Method* and *Disperse Coupling–Long Method* co-occurrences.

Additionally, the *Feature Envy–Long Method* and *Feature Envy–God Class* co-occurrences should also receive special attention from the developers as they are second in the Table 10 and in the Section 4.1 they also occurred with a certain frequency since they were in 2 of the 3 systems studied in this work. We were also able to get some interesting explanations as to why developers have refactored a certain code smells co-occurrence:

P1:"I always considered the smell easier to refactor in each co-occurrence and the difficulty in refactoring that smell as being the difficulty in refactoring the co-occurrence."

P3: "I chose the difficulties based on the difficulty of each code smell involved. So the cases that have God Class and Shotgun Surgery are the ones that have the greatest difficulties and the easiest are those that have Feature envy, Long method and Refused Parent Bequest."

P4: "The fact that god class is the most complex due to the fact that we have to create another class and have not refactored it yet, and the long method is the easiest because we already had experience."

P5:"Due to the fact that to correct the co-occurrence it was necessary to correct the Feature Envy" code-smell, which did not require many changes."

We found that developers had more difficulty and did not like to remove code smells like God Class, Shotgun Surgery, Disperse Coupling and Intensive Coupling from co-occurrences. According to the developers, these smells "*involve a lot of functions and variables,*" "*require changes in more places in the code,*" and "*require a lot of operations to remove them.*" While the smells like Feature Envy, Long Method and Refused Parent Bequest were easier to refactor because, according to the developers, *"refactorings are simpler and more quickly*" and "*it was necessary to refactor one or a few methods.*"

These responses indicate that the developers did not consider co-occurrence as a whole, but what were the code smells that were part of that co-occurrence.

5 THREATS TO VALIDITY

This section discusses threats to validity of the study according to the classification of Wohlin et al. [53].

Internal validity. An internal threat of this study is the low number of systems analyzed. However, the systems used in this study are closed-source systems, which are poorly analyzed in the literature. Another issue identified is that the analyzed classes are production entities, or that is, we do not consider the test entities used to test the production classes when measuring internal quality attributes. However, we consider that developers are more concerned with classes that provide some features to the system and not test classes.

Construct validity. Code smells were automatically identified by the JSpIRIT and JDeodorant tools, reducing the chance of errors in detection. Even so, the strategies implemented by these tools can be a potential threat to validity. In this way, other detection tools could use different strategies than the tools used in this study. Thus, this could cause a variation in the set of identified code smells and consequently affect the detection of co-occurrences of code smells. There are several types of code smells relationships found in other studies, such as coupled smells and colocated smells [54]. In our study, we consider co-occurrences of smells to occur at the class and method levels.

External validity. The results can only be used for object-oriented systems written in Java. A limitation is the domain of systems. From other domains it is possible to have different results. Another problem that we have identified is that there are developers with little development experience or little knowledge of code smells, refactoring or quality metrics. To mitigate this problem, we conducted training with all developers.

6 RELATED WORK

Kaur [23] conducted a systematic literature review of such existing empirical studies that investigate the impact of code smells on software quality attributes. The results of this study [23] indicate that the impact of code smells on software quality is not uniform as different code smells have the opposite effect on different software quality attributes. Similar to the conclusions of the studies identified in the systematic review, in our work, we also identified that the removal of code smells co-occurrences also does not have a uniform impact on internal quality attributes. de Paulo Sobrinho et al. [12] and Lacerda et al. [26] perform a literature review on code smells. Both papers [12, 26] find that code smells God Class, Feature Envy, Long Method are the most studied in the context of co-occurrences and that code smells such as Refused Parent Bequest and Shotgun Surgery need more attention. Also, the results found in both studies indicate that the presence of code smells in the source code may report problems in maintainability and software design. However, the authors point out that this is an area that still needs attention that further empirical studies are needed on the impact of these anomalies on the source code. In our research, we took into account the Shotgun Surgery and Refused Parent Bequest code smells and found that removing code smells co-occurrences may indicate a decrease in the complexity attribute.

Abbes et al. [1] studies the interactions between code smells and their effects. The authors concluded that when code smells appeared isolated, they had no impact on maintainability, but when they appeared interconnected, they brought a major maintenance effort. Fernandes et al. [15] extends a large quantitative study about the refactoring effect on internal quality attributes with new insights. As a result of this study, the authors identified that most refactoring types improved one or more attributes, and re-refactoring affects the internal quality attributes similarly to refactoring in general. In our study, we identified that by removing code smells co-occurrences, there is a reduction in complexity. Still, we did not see a positive impact on other attributes such as cohesion and coupling.

Yamashita and Moonen [55] analyzes the impact of the inter-Smell relations in the maintainability of four medium-sized industrial systems written in Java. The authors detect significant relationships between Feature Envy, God Class and Long Method and conclude that Inter-Smell relationships are associated with problems during maintenance activities. Palomba et al. [39] conducted a large-scale empirical study aimed at quantifying the diffuseness of the problem in terms of how frequently code smells occur together. As a result, the authors identify that 59% of smelly classes are affected by more than one smell. In particular, six pairs of smell types frequently co-occur Message Chains–Spaghetti Code, Message Chains–Complex Class, Message Chains–Blob, Message Chains–Refused Bequest, Long Method–Spaghetti Code and Long Method–Feature Envy. In our study, we identified that the most frequent code smells co-occurrences are: God Class-Long Method and Disperse Coupling-Long Method. In these studies, we found that the most frequent co-occurrences are different. Thus, more studies are needed to analyze how these co-occurrences are inserted in different types of systems.

The studies reinforce that developers must be alerted about the impacts that code smells co-occurrences in the code so that these relationships are refactored and removed in the initial stages of implementation. However, we have not identified studies that investigate the impact of refactorings of code smell co-occurrences on internal quality attributes.

7 CONCLUSION AND FUTURE WORK

Our study considered 7 types of code smells and their co-occurrences in 3 Java OO closed-source systems, and 5 internal quality attributes (cohesion, inheritance, size, coupling, and complexity). As the main objective of our study: (i) we analyzed the co-occurrences that most tend to co-occur in these systems; (ii) we investigated the impact of removing these anomalies for internal quality attributes; and, (iii) we identified which are the co-occurrences to be removed according to the developers' perspective. The process of removing these co-occurrences of code smells took 3 months and happened at different times for each system, a total of 183 commits were made and 94 co-occurrences were removed.

The main findings of our study were: (i) *God Class–Long Method* and *Disperse Coupling–Long Method* are the most frequent co-occurrences in the three systems and also the most difficult co-occurrences to refactor in developers' perspective; (ii) co-occurrences increase during the development of the system; (iii) the removal of these anomalies has a negative impact on the cohesion and coupling attributes; and (iv) the removal of code smells co-occurrences suggests a significant decrease in the complexity of the systems. The finding (iv) is interesting because several studies in the literature point out the harms of high complexity for OO systems [10, 35, 45]. As future work, we want to: (i) analyze more closed-source systems and also analyze open-source systems; (ii) reproduction of the study with tools that detect other code smells; and, (iii) verify which are the most harmful co-occurrences for the internal quality attributes.

ACKNOWLEDGMENTS

This work was partially funded by CNPq (434969/2018-4, 312149/2016-6, 141285/2019-2), CAPES/Procad (175956), and FAPERJ (200773/2019, 010002285/2019).

REFERENCES

[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2011. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *15th CSMR*. IEEE, 181–190.

[2] Jehad Al Dallal. 2013. Object-oriented class maintainability prediction using internal quality attributes. *Inf. Softw. Technol.* 55, 11 (2013), 2028–2048.

[3] Rafa E Al-Qutaish. 2010. Quality models in software engineering literature: an analytical and comparative study. *Journal of American Science* 6, 3 (2010), 166–175.

[4] Mamdouh Alenezi and Khaled Almustafa. 2015. Empirical analysis of the complexity evolution in open-source software systems. *International Journal of Hybrid Information Technology* 8, 2 (2015), 257–266.

[5] Ajay Bandi, Byron J Williams, and Edward B Allen. 2013. Empirical evidence of code decay: A systematic mapping study. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 341–350.

[6] James M Bieman and Byung-Kyoo Kang. 1995. Cohesion and reuse in an object-oriented system. *ACM SIGSOFT Software Engineering Notes* 20, SI (1995), 259–262.

[7] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How does refactoring affect internal quality attributes?: A multi-project study. In *31st SBES*. ACM, 74–83.

[8] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.* 20, 6 (1994), 476–493.

[9] Istehad Chowdhury and Mohammad Zulkernine. 2010. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?. In *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 1963–1969.

[10] David P Darcy, Chris F Kemerer, Sandra A Slaughter, and James E Tomayko. 2005. The structural complexity of software an experimental test. *IEEE Trans. Softw. Eng.* 31, 11 (2005), 982–995.

[11] Rafael de Mello, Anderson Uchôa, Roberto Oliveira, Willian Oizumi, Jairo Souza, Kleyson Mendes, Daniel Oliveira, Baldoino Fonseca, and Alessandro Garcia. 2019. Do Research and Practice of Code Smell Identification Walk Together? A Social Representations Analysis. In *13th ESEM*. 1–6.

[12] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. 2018. A systematic literature review on bad smells—5 W's: which, when, what, who, where. *IEEE Trans. Softw. Eng.* (2018).

[13] Giuseppe Destefanis, Steve Counsell, Giulio Concas, and Roberto Tonelli. 2014. Software metrics in agile software: An empirical study. In *International Conference on Agile Software Development*. Springer, 157–170.

[14] Robert Dyer, Hridesh Rajan, and Yuanfang Cai. 2012. An exploratory study of the design impact of language features for aspect-oriented interfaces. In *11th AOSD*. ACM, 143–154.

[15] Eduardo Fernandes, Alexander Chávez, Alessandro Garcia, Isabella Ferreira, Diego Cedrim, Leonardo Sousa, and Willian Oizumi. 2020. Refactoring Effect on Internal Quality Attributes: What Haven't They Told You Yet? *Inf. Softw. Technol.* (2020), 106347.

[16] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A review-based comparative study of bad smell detection tools. In *20th EASE*. ACM, 18.

[17] Eduardo Fernandes, Gustavo Vale, Leonardo Sousa, Eduardo Figueiredo, Alessandro Garcia, and Jaejoon Lee. 2017. No Code Anomaly is an Island. In *16th ICSR*. Springer, 48–64.

[18] Francesca Arcelli Fontana, Vincenzo Ferme, and Marco Zanoni. 2015. Towards assessing software architecture quality by exploiting code smell relations. In *2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics*. IEEE, 1–7.

[19] Francesca Arcelli Fontana, Valentina Lenarduzzi, Riccardo Roveda, and Davide Taibi. 2019. Are architectural smells independent from code smells? An empirical study. *J. Syst. Softw.* 154 (2019), 139–156.

[20] Martin Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[21] ISO. 2011. IEC 25010: 2011 systems and software engineering–systems and software quality requirements and evaluation (square)–system and software quality models. *International Organization for Standardization* 34 (2011), 2910.

[22] Fehmi Jaafar, Angela Lozano, Yann-Gaël Guéhéneuc, and Kim Mens. 2017. On the analysis of co-occurrence of anti-patterns and clones. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 274–284.

[23] Amandeep Kaur. 2019. A Systematic Literature Review on Empirical Analysis of the Relationship Between Code Smells and Software Quality Attributes. *Archives of Computational Methods in Engineering* (2019), 1–30.

[24] Sharanpreet Kaur and Satwinder Singh. 2016. Spotting & eliminating type checking code smells using eclipse plug-in: Jdeodorant. *International Journal of Computer Science and Communication Engineering* 5, 1 (2016).

[25] V Krishnapriya and K Ramar. 2010. Exploring the difference between object oriented class inheritance and interfaces using coupling measures. In *2010 International Conference on Advances in Computer Engineering*. IEEE, 207–211.

[26] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: a tertiary systematic review of challenges and observations. *J. Syst. Softw.* (2020), 110610.

[27] Michele Lanza and Radu Marinescu. 2007. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media.

[28] Mark Lorenz and Jeff Kidd. 1994. *Object-oriented software metrics: a practical guide.* Prentice-Hall, Inc.

[29] Angela Lozano, Kim Mens, and Jawira Portugal. 2015. Analyzing code evolution to uncover relations. In *2nd PPAP*. IEEE, 1–4.

[30] Isela Macia, Joshua Garcia, Daniel Popescu, Alessandro Garcia, Nenad Medvidovic, and Arndt von Staa. 2012. Are automatically-detected code anomalies relevant to architectural modularity?. In *11th AOSD*. 167–178.

[31] Ruchika Malhotra and Anuradha Chug. 2016. An empirical study to assess the effects of refactoring on software maintainability. In *International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 110–117.

[32] Júlio Martins, Carla Ilane Moreira Bezerra, and Anderson Uchôa. 2019. Analyzing the Impact of Inter-smell Relations on Software Maintainability: An Empirical Study with Software Product Lines. In *Proceedings of the XV Brazilian Symposium on Information Systems*. 1–8.

[33] Thomas J McCabe. 1976. A complexity measure. *IEEE Trans. Softw. Eng.* 4 (1976), 308–320.

[34] Sandro Morasca. 2009. A probability-based approach for measuring external attributes of software artifacts. In *3rd ESEM*. IEEE Computer Society, 44–55.

[35] Iulian Neamtiu, Guowu Xie, and Jianbo Chen. 2013. Towards a better understanding of software evolution: an empirical study on open-source software. *J. Softw.: Evol. Process* 25, 3 (2013), 193–218.

[36] Willian Oizumi, Alessandro Garcia, Leonardo da Silva Sousa, Bruno Cafeo, and Yixue Zhao. 2016. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *38th ICSE*. IEEE, 440–451.

[37] Willian Nalepa Oizumi, Alessandro Fabricio Garcia, Thelma Elita Colanzi, Manuele Ferreira, and Arndt von Staa. 2014. When code-anomaly agglomerations represent architectural problems? An exploratory study. In *28th SBES*. IEEE, 91–100.

[38] Matheus Paixão, Anderson Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. 2020. Behind the Intents: An In-depth Empirical Study on Software Refactoring in Modern Code Review. In *17th MSR*.

[39] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. A large-scale empirical study on the lifecycle of code smell co-occurrences. *Inf. Softw. Technol.* 99 (2018), 1–10.

[40] Fabio Palomba, Rocco Oliveto, and Andrea De Lucia. 2017. Investigating code smell co-occurrences using association rule learning: A replicated study. In *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 8–13.

[41] Błażej Pietrzak and Bartosz Walter. 2006. Leveraging code smell detection with inter-smell relations. *Extreme Programming and Agile Processes in Software Engineering* (2006), 75–84.

[42] Benjamin Reilly. 2002. Social choice in the south seas: Electoral innovation and the borda count in the pacific island countries. *International Political Science Review* 23, 4 (2002), 355–372.

[43] José Amancio M Santos, João B Rocha-Junior, Luciana Carla Lins Prates, Rogeres Santos do Nascimento, Mydiã Falcão Freitas, and Manoel Gomes de Mendonça. 2018. A systematic review on the code smell effect. *J. Syst. Softw.* 144 (2018), 450–477.

[44] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. 2012. Quantifying the effect of code smells on maintenance effort. *IEEE Trans. Softw. Eng.* 39, 8 (2012), 1144–1156.

[45] Ramanath Subramanyam and Mayuram S. Krishnan. 2003. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.* 29, 4 (2003), 297–310.

[46] Sandhya Tarwani and Anuradha Chug. 2016. Sequencing of refactoring techniques by Greedy algorithm for maximizing maintainability. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 1397–1403.

[47] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2018. Ten years of JDeodorant: Lessons learned from the hunt for smells. In *25th SANER*. IEEE, 4–14.

[48] Anderson Uchôa, Eduardo Fernandes, Ana Carla Bibiano, and Alessandro Garcia. 2017. Do Coupling Metrics Help Characterize Critical Components in Component-based SPL? An Empirical Study. In *Proceedings of the 5th Workshop on Software Visualization, Evolution and Maintenance (VEM)*. 36–43.

[49] Anderson Uchôa, Caio Barbosa, Willian Oizumi, Publio Blenílio, Rafael Lima, Alessandro Garcia, and Carla Bezerra. 2020. How Does Modern Code Review Impact Software Design Degradation? An In-depth Empirical Study. In *36th ICSME*. 1 – 12.

[50] Santiago Vidal, Hernan Vazquez, J Andres Diaz-Pace, Claudia Marcos, Alessandro Garcia, and Willian Oizumi. 2015. JSpIRIT: a flexible tool for the analysis of code smells. In *34th SCCC*. IEEE, 1–6.

[51] Santiago A Vidal, Claudia Marcos, and J Andrés Díaz-Pace. 2016. An approach to prioritize code smells for refactoring. *Automated Software Engineering* 23, 3 (2016), 501–532.

[52] Bartosz Walter, Francesca Arcelli Fontana, and Vincenzo Ferme. 2018. Code smells and their collocations: A large-scale experiment on open-source systems. *J. Syst. Softw.* 144 (2018), 1–21.

[53] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering.* Springer Science & Business Media.

[54] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey. In *20th WCRE*. IEEE, 242–251.

[55] Aiko Yamashita and Leon Moonen. 2013. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *35th ICSE*. IEEE, 682–691.

[56] Aiko Yamashita, Marco Zanoni, Francesca Arcelli Fontana, and Bartosz Walter. 2015. Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In *31st ICSME*. IEEE, 121–130.